Attorney Docket No.
225671
MS # 307237.01

PATENT APPLICATION

Invention Title:

METHOD AND SYSTEM FOR DETECTING POTENTIAL RACES IN MULTITHREADED PROGRAMS

Inventors:

Yuan Yu	US	Cupetino	California
INVENTOR'S NAME	CITIZENSHIP	CITY OF RESIDENCE	STATE or FOREIGN COUNTRY
INVENTOR'S NAME	CITIZENSHIP	CITY OF RESIDENCE	STATE or FOREIGN COUNTRY
INVENTOR'S NAME	CITIZENSHIP	CITY OF RESIDENCE	STATE or FOREIGN COUNTRY
INVENTOR'S NAME	CITIZENSHIP	CITY OF RESIDENCE	STATE or FOREIGN COUNTRY
INVENTOR'S NAME	CITIZENSHIP	CITY OF RESIDENCE	STATE or FOREIGN COUNTRY
INVENTOR'S NAME	CITIZENSHIP	CITY OF RESIDENCE	STATE or FOREIGN COUNTRY

Be it known that the inventors listed above have invented a certain new and useful invention with the title shown above of which the following is a specification.

METHOD AND SYSTEM FOR DETECTING POTENTIAL RACES IN MULTITHREADED PROGRAMS

TECHNICAL FIELD OF THE INVENTION

[0001] The present invention relates generally to computer systems, and more particularly to detecting race conditions in multithreaded computer programs.

BACKGROUND OF THE INVENTION

[0002] It has become common for computer software developers to write programs making use of multiple threads of execution. Modern operating systems and programming languages support threads, and many large commercial applications are multithreaded. Threads are especially useful for implementing multiple asynchronous computations within an operating system process. Event-driven applications, for example, often employ multithreading.

[0003] The very features that make multithreading a useful programming technique also make debugging multithreaded programs a very difficult task, however. Multiple threads can interact in nondeterministic and timing-dependent ways. Typically such threads share data, requiring synchronization of their interaction to ensure program correctness, independent of how threads are scheduled or how their instruction streams are interleaved.

[0004] It is particularly difficult for programmers to detect errors in thread synchronization that are associated with race conditions. In a multithreaded program, a data race condition occurs when a shared memory location is accessed by two or more concurrent threads, with at least one of the accesses being a write, without proper synchronization to constrain the ordering of the accesses. The effects of the execution in such a case depend on the particular order in which the accesses take place. Race conditions often result in unexpected and undesirable program behavior, such as program crashes or incorrect results. Such nondeterminacy is also precisely why it is so difficult to detect race conditions using conventional debugging techniques.

[0005] Given the potentially detrimental effects of race conditions and the difficulty of debugging programs that contain them, automated tools for detecting the presence of race conditions should be of great value to developers of multithreaded programs. Effective and efficient tools have been lacking, however. With respect to dynamic race detection, in which an attempt is made to detect potential races in a particular execution of a program, two approaches have been widely used: the Lamport "happens-before" order and the lockset technique, which are described further in the detailed description below. The former typically has very unsatisfactory runtime overhead, especially for programs written in object-oriented languages like C# and Java, while the latter approach often produces an

unacceptable number of false positives, particularly in programs using asynchronous delegates.

SUMMARY OF THE INVENTION

[0006] The following presents a simplified summary of some embodiments of the invention in order to provide a basic understanding of the invention. This summary is not an extensive overview of the invention. It is not intended to identify key or critical elements of the invention or to delineate the scope of the invention. Its sole purpose is to present some embodiments of the invention in simplified form as a prelude to the more detailed description that is presented below.

[0007] In accordance with one embodiment of the invention, a system for dynamic race detection is provided. The system includes a mechanism for maintaining a set of concurrent thread segments that access a shared memory location; a mechanism for maintaining, with respect to a running thread, a set of thread segments that are ordered before its current thread segment; a mechanism for maintaining a first set of locks associated with a shared memory location; a mechanism for maintaining a second set of locks associated with a thread that acquires and releases the locks in the second set of locks; and a mechanism for reporting a detected race condition.

[0008] In accordance with another embodiment, a method for dynamic race detection is provided. The method includes (a) maintaining a first set of locks associated with a shared memory location; (b) maintaining a second set of locks associated with a thread that acquires and releases the locks in the second set of locks; (c) maintaining a set of concurrent thread segments that access a shared memory location; and (d) maintaining, with respect to a thread, a set of thread segments that are ordered before the current segment of the thread.

[0009] Both the set of concurrent thread segments that access a shared memory location and the set of thread segments that are ordered before a thread may be represented as a set of ordered pairs, wherein one member of a pair in the set of ordered pairs is a thread identifier, and the other member of a pair is a virtual clock value identifying a thread segment of the first member.

[0010] Each thread maintains a virtual clock that is initialized to zero at the thread creation and is incremented by one whenever the thread forks another thread. When a thread forks a second thread, the set of thread segments that are ordered before the second thread is computed as the set union of (i) the set of thread segments that are ordered before the first thread and (ii) a singleton set comprising the thread segment of the first thread at which the second thread is forked. In one embodiment, the virtual clock associated with the first thread is incremented by one, and the virtual clock associated with the forked thread is initialized to zero. When a

thread performs a join operation on a second thread, the set of thread segments that are ordered before the first thread is computed as the union of (i) the set of thread segments that are ordered before the first thread, (ii) a subset of the set of thread segments that are ordered before the second thread wherein, for each thread segment in the subset, the thread identifier of the thread segment is not equal to the thread identifier of the first thread, and (iii) the singleton set containing the current thread segment of the second thread.

[0011] If a thread accesses a shared memory location, the set of concurrent thread segments accessing the location is updated by removing thread segments that are no longer concurrently accessing the location, and adding the current thread segment of the thread. If the new set of concurrent thread segments contains no more than one element, then the set of locks associated with the shared memory location is updated to the set of locks associated with the thread, and otherwise is updated to the intersection of the set of locks associated with the shared memory location and the set of locks associated with the thread. If the set of concurrent thread segments has more than one element, and the set of locks associated with the shared memory location is empty, a warning of a potential race condition is reported.

[0012] In accordance with another embodiment, a dynamic race detection system and method are provided. In a runtime system, calls to a race detector are emitted when code in the form of a common intermediate language is loaded and

compiled in the execution engine. The data structure for storing instrumentation information needed for a memory object is allocated together with the object by the memory allocation mechanism of the runtime system.

[0013] Other features of the invention will become apparent from the following detailed description when taken in conjunction with the drawings, in which:

BRIEF DESCRIPTION OF THE DRAWINGS

[0014] FIG. 1 is a diagram representing a happens-before order of events associated with two concurrent threads in accordance with the prior art;

[0015] FIG. 2A is a flowchart illustrating the maintenance of the lockset associated with a thread in accordance with the prior art;

[0016] FIG. 2B is a flowchart illustrating the maintenance of the lockset associated with a shared memory location in accordance with the prior art;

[0017] FIG. 3 is a diagram illustrating the operation of the lockset approach on an example with no race condition in accordance with the prior art;

[0018] FIG. 4 is a diagram illustrating the operation of the lockset approach on an example with a correctly-reported race condition in accordance with the prior art;

[0019] FIG. 5 is a diagram representing the operation of a thread performing a fork and join, giving rise to a false positive under the lockset approach in accordance with the prior art;

[0020] FIG. 6 is a diagram representing the operation of a thread performing a fork and join, illustrating the maintenance of the thread's virtual clock in accordance with an embodiment of the invention;

[0021] FIG. 7A is a flowchart showing steps associated with a fork call in accordance with an embodiment of the invention;

[0022] FIG. 7B is a flowchart showing steps associated with a join call in accordance with an embodiment of the invention;

[0023] FIG. 8 is a flowchart showing steps associated with a read or a write of a memory location in accordance with an embodiment of the invention; and

[0024] FIG. 9 is a diagram representing the operation of a thread performing a fork and join, in which a race condition is correctly not reported for a single-threaded access of a memory location, in accordance with an embodiment of the invention.

DETAILED DESCRIPTION

[0025] In the following description, various embodiments of the present invention will be described. For purposes of explanation, specific configurations and details are set forth in order to provide a thorough understanding of the embodiments. However, it will also be apparent to those having skill in the art that the present invention may be practiced without the specific details. Furthermore, well-known features may be omitted or simplified in order not to obscure the embodiment being described.

[0026] Before proceeding with a description of the invention, the happens-before and lockset approaches in the prior art, mentioned in the background section above, will be described in some detail in order to elucidate the novelty and utility of the present invention. In the happens-before approach, a partial order of all events associated with all threads in a concurrent execution is created. The order is based on the relation described in Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," 21 Commun. ACM 558-565 (1978), incorporated herein by reference. Within a single thread, events are ordered in the order in which they occur. Between threads, events are ordered according to the properties of the locks or other synchronization objects that are acquired and released by the threads. If one thread accesses a lock, and the next access of the lock is by a different thread, the first access is defined to "happen before" the second if the semantics of the lock prevent a

schedule in which the two events are exchanged in time. A race is deemed to have occurred if two threads access a shared memory location and the accesses are causally unordered.

[0027] As a simple illustration of the happens-before approach, consider two concurrent threads t_1 and t_2, each of which executes the following code fragment:

```
acquire(l);   // Acquire lock l
write x;      // Write shared location x
release(l);   // Release lock l
```

[0028] FIG. 1 shows a possible ordering of the events associated with the two threads t_1 101 and t_2 103. The three program statements 105, 107, 109 executed by t_1 101 are ordered by happens-before because they are executed sequentially in the same thread. The acquire of lock l by t_2 103 is ordered by happens-before with the release of lock l by t_1 101 because a particular lock cannot be acquired before it is released by its previous holder. Finally, the three statements 111, 113, 115 executed by t_2 103 are ordered by happens-before because they are executed sequentially within that thread.

[0029] Lockset-based detection is described in a number of references, such as Savage et al., "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," 15 ACM Trans. Comp. Sys. 391-411 (1997), incorporated herein by reference. In a simple lockset approach, for each shared location x, a set S_x of locks that protect x is

maintained for the computation by monitoring all reads and writes to x as the program executes. For each thread t, a set S_t of locks held by t is maintained by monitoring lock acquisitions of the thread t.

[0030] The flow diagrams of FIGS. 2A and 2B illustrate how the sets S_t and S_x, respectively, are maintained under the lockset approach. Turning to FIG. 2A, at step 201, S_t is initially empty when the thread t is created. At step 203, it is determined whether the thread t acquires a lock l. If so, at step 205 S_t is updated by taking the union of S_t and the singleton set containing l. Similarly, at step 207, it is determined whether the thread releases a lock l. If so, at step 209 S_t is updated by taking the set difference of S_t and the singleton set containing l, thus removing l from the set of locks held by thread t. Turning now to FIG. 2B, at step 211, initially S_x is L, the set of all possible locks. At step 213, it is determined whether the thread t performs a read or write operation on location x. If so, at step 215 S_x is updated to the set comprising the intersection of S_x and S_t. At step 217, it is determined whether S_x is empty. If so, at step 219 a warning regarding a potential race condition is reported.

[0031] FIG. 3 illustrates a simple example of the operation of the lockset approach, using the same exemplary concurrent threads t_1 and t_2 as in FIG. 1.

Suppose that threads t_1 and t_2 execute their respective statements in the following order:

	t_1		t_2
1	`acquire(l);`	4	`acquire(l);`
2	`write x;`	5	`write x;`
3	`release(l);`	6	`release(l);`

That is, thread t_1 executes its three statements before thread t_2 executes its three statements. We also assume that S_{t_1} and S_{t_2} are empty and S_x is L at the start of the execution.

[0032] Turning now to FIG. 3, the flow diagram represents the sequence of statements executed by threads t_1 301 and t_2 303. Block 305 shows that initially S_{t_1} is empty, and after t_1 301 acquires lock l, S_{t_1} becomes the set containing l. In block 307, t_1 301 executes the write of x, following which S_x also becomes the set containing l. In block 309, t_1 301 releases the lock l, and S_{t_1} again becomes the empty set. Now thread t_1 301 has ended its execution and thread t_2 303 begins. Block 311 shows that initially S_{t_2} is empty, and after t_2 303 acquires lock l, S_{t_2} becomes the set containing l. In block 313, t_2 303 writes location x, following which S_x becomes the set formed by the intersection of S_x and S_{t_2}. Since both sets are now the singleton set containing l, S_x remains the set containing l. In block 315, t_2 303 releases the lock l,

and S_{t_2} again is empty. During the execution of threads t_1 301 and t_2 302, S_x is never empty, so the lockset method does not report any race condition. Indeed, the lack of a race condition intuitively follows from the fact that a single lock l is used to protect x in both threads.

[0033] FIG. 4 provides a second example of the operation of the lockset approach. Here thread t_1 401 and thread t_2 403 use different locks, l_1 and l_2 respectively, unlike the previous example in which x was protected by the same lock in both threads. Block 405 shows that initially S_{t_1} is empty, and after t_1 401 acquires lock l_1, S_{t_1} becomes the set containing l_1. In block 407, t_1 401 executes the write of x, following which S_x also becomes the set containing l_1. In block 409, t_1 301 releases the lock l_1, and S_{t_1} again becomes the empty set. Now thread t_1 401 has ended its execution and thread t_2 403 begins. Block 411 shows that initially S_{t_2} is empty, and after t_2 403 acquires lock l_2, S_{t_2} becomes the set containing l_2. In block 413, t_2 403 writes location x, following which S_x becomes the set formed by the intersection of S_x and S_{t_2}. S_x is the set containing l_1, and S_{t_2} is the set containing l_2, so their intersection is the empty set. Since S_x is empty, a race condition is reported at this point during the execution. In block 415, t_2 403 releases the lock l_2, and S_{t_2} again is empty.

[0034] While tools implementing the lockset approach do not generally have the poor performance associated with approaches that use the happens-before relation, such tools are known to produce many false positives, reporting race conditions for race-free programs. The most common class of false positives comprise those which arise when threads make fork and join (wait) system calls. An example is shown in FIG. 5. Let us assume that there is only one thread t at the start of the execution. The vertical line 501 represents the execution of thread t. The diagonal line 503 extending from the line 501 represents t forking a new thread t_1. The vertical line 505 represents the execution of thread t_1. After the fork, there are now two threads, t and t_1, executing concurrently. The diagonal line 507 extending towards the line 501 represents thread t joining thread t_1: that is, thread t waits for t_1 to complete execution.

[0035] The fork 503 and join 507 implicitly impose an ordering on the events in threads t 501 and t_1 505. In block 509, thread t executes a write of x, with an acquire and release of a lock l. In block 511, thread t_1 executes the same statements. The two accesses of x do not give rise to a race condition, because they are protected by the same lock l. At block 513, thread t executes a write of x. At this point thread t can write x without having to use a lock, and without having to use the same lock used to protect x in the execution by thread t_2. As indicated in block 513, however, before

the write of x, S_x is the set containing l. After the write of x, S_x becomes empty. Because S_x is empty, the lockset procedure reports a race, even though there clearly is no race, for only a single thread is being executed and there is no need to protect x from access.

[0036] The present invention extends the lockset approach to eliminate its tendency to report false positives in the fork and join context. In addition to recording the per-thread and per-location locksets as in the lockset approach, the invention maintains two further sets. One set is the set T_x, comprising the set of concurrent thread segments accessing the shared memory location x. A race is reported when the relevant lockset is empty and the cardinality of T_x is greater than 1. The second new set is the set B_t, comprising the set of thread segments ordered before the current thread segment of t. In one embodiment, both T_x and B_t are represented as the set of tuples $\{<t_1, c_1>, \ldots, <t_n, c_n>\}$. The ordering relies on the use of a virtual clock C_t for each thread t. A tuple $<t, c>$ represents the thread segment of thread t at the virtual clock time of c.

[0037] FIG. 6 is a diagram representing a fork and join, illustrating the manner in which virtual clocks are maintained for each thread in an embodiment of the invention. The vertical line 607 represents the execution of thread t. Let us assume that the virtual clock for t is initially 0 (at the point designated 609). The diagonal

line 611 represents t forking the thread t_1. At this point on the execution line 607 for thread t (designated 613), the virtual clock for t is incremented by 1. The vertical line 615 represents the execution of the forked thread t_1. The virtual clock for thread t_1 is initialized to 0 (at the point designated 617). The diagonal line 619 represents t's join of t_1.

[0038] The flowchart of FIG. 7A shows steps associated with the fork of a new concurrent thread t_1 by a thread t. At step 703, it is determined whether thread t forks thread t_1. If so, the following operations are performed. At step 703 the set B_{t_1}, the set of thread segments ordered before thread t_1, is computed as the union of the corresponding set for thread t and the singleton set comprising the thread segment $<t, C_t>$. At step 705 the virtual clock for thread t is incremented by 1. At step 707 the virtual clock for the newly forked thread t_1 is initialized to 0. The flowchart of FIG. 7B shows steps associated with the join by thread t. At step 711, it is determined whether thread t makes the join call, waiting for thread t_1 to complete execution. If so, at step 713, the new value of B_t is computed as the union of the following three sets: (i) B_t, (ii) the set of thread segments in set B_{t_1} that do not belong to the current thread t, and (iii) the singleton set containing the current thread segment of t_1.

[0039] The steps illustrated in FIG. 2A and discussed above, regarding the maintenance of the set S_t as locks are acquired and released, are also used in the present invention and need not be described further here.

[0040] The flowchart of FIG. 8 shows steps taken when a read or a write of location x is executed by thread t. At step 801 the new value of set T_x is computed, representing the thread segments concurrently accessing location x after the read or write. The second part of the union forming the new T_x is the set containing t's current thread segment $<t, C_t>$. Since thread t is reading or writing x, clearly t is one of the threads that should be in the set T_x. The first part of the union represents a subset of the old value of set T_x in which any thread that is no longer concurrently accessing the location is filtered out. At step 803 it is determined whether the cardinality of the new T_x is less than or equal to 1. If so, there is at most one thread currently accessing x. The new value of S_x then becomes the current value of the lockset S_t (step 805). Otherwise, at step 807, there are multiple concurrent threads accessing x, and the new value of S_x becomes the set comprising the intersection of the old value of S_x and S_t. At step 809 it is determined whether (a) the new value of S_x is empty and (b) the cardinality of the new value of T_x is greater than 1. If so, at step 811 a potential race condition is reported.

[0041] Turning now to FIG. 9, a diagram of a fork and join similar to FIG. 5 is shown, but in which race detection follows the approach of the present invention. Let us assume that, at the start of the execution (block 901), the sets T_x and B_t are empty, and S_x is the singleton set containing the lock l. Suppose that block 905 is executed before block 907.

[0042] After the execution of block 905, S_x is the set containing l, and T_x is set containing the thread segment <t, 1>. After the execution of block 907, S_x remains the same, but T_x now contains two thread segments <t, 1> and <$t1$, 0>. Following the join at block 909, B_t becomes the set containing the thread segment <$t1$, 0>. Before the write of x by t, at block 911, S_x is the set containing l, and T_x is still the set containing the two thread segments <t, 1> and <$t1$, 0>. Following the write, S_x becomes empty, and T_x becomes the set containing the thread segment <t, 1> (t is the only thread that is concurrently accessing x). Since the cardinality of T_x is 1, a race condition is, correctly, not reported in accordance with the present invention.

[0043] In the prior art, there are two widely used approaches in implementing the kind of race detector included in embodiments of the present invention as described in this specification. The first approach is to insert calls for the memory accesses at the source code or byte code level. The second approach is to insert calls for load and store instructions in the native code. The previous approaches make the

race detection tool tedious to run when there are a large number of shared libraries to be dynamically linked in, and, more significantly, this normally entails high runtime cost. In certain embodiments of the present invention, the dynamic race detector is instead implemented within a runtime system.

[0044] In one embodiment, the race detector is implemented within the Common Language Runtime of the .NET framework of Microsoft Corporation. The JIT (Just-In-Time) compiler, which compiles byte code to native code, is modified so that when code is dynamically loaded and compiled, calls to the race detector are inserted. The mechanism for allocating shared memory objects is modified so that the allocation adds on to the object the instrumentation information needed for the race detector; the information is then automatically managed by the runtime's garbage collector.

[0045] This implementation approach has a number of advantages over previous techniques. First, all the code being executed is dynamically instrumented. Second, modifying the JIT rather than the byte code avoids certain problems with the byte code verification phase. Third, having the JIT insert calls to the race detector permits the use of information gathered by the compiler to optimize the instrumentation. For example, the compiler may determine that instrumentation is not necessary, as for example if the compiler determines that a field is declared read-only or is local to a particular thread and invisible to other threads. Fourth, the race

detector interacts well with the garbage collector, instrumenting only the shared objects on the managed heap and preventing potential memory leak problems caused by the race detection instrumentation. Finally, the implementation provides a performance advantage in that it eliminates a level of indirection present in standard approaches to race detector implementation.

[0046] Other variations are within the spirit of the present invention. Thus, while the invention is susceptible to various modifications and alternative constructions, a certain illustrated embodiment thereof is shown in the drawings and has been described above in detail. It should be understood, however, that there is no intention to limit the invention to the specific form or forms disclosed, but on the contrary, the intention is to cover all modifications, alternative constructions, and equivalents falling within the spirit and scope of the invention, as defined in the appended claims.

[0047] All methods described herein can be performed in any suitable order unless otherwise indicated herein or otherwise clearly contradicted by context. The use of any and all examples, or exemplary language (e.g., "such as") provided herein, is intended merely to better illuminate embodiments of the invention and does not pose a limitation on the scope of the invention unless otherwise claimed. No language in the specification should be construed as indicating any non-claimed element as essential to the practice of the invention.

[0048] Preferred embodiments of this invention are described herein, including the best mode known to the inventor for carrying out the invention. Variations of those preferred embodiments may become apparent to those of ordinary skill in the art upon reading the foregoing description. The inventor expects skilled artisans to employ such variations as appropriate, and the inventor intends for the invention to be practiced otherwise than as specifically described herein. Accordingly, this invention includes all modifications and equivalents of the subject matter recited in the claims appended hereto as permitted by applicable law. Moreover, any combination of the above-described elements in all possible variations thereof is encompassed by the invention unless otherwise indicated herein or otherwise clearly contradicted by context.